1 INTRODUCTION

Jealous stepmother and sisters; magical aid by a beast; a marriage won by gifts magically provided; a bird revealing a secret; a recognition by aid of a ring; or show; or what not; a dénouement of punishment; a happy marriage - all those things, which in sequence, make up Cinderella, may and do occur in an incalculable number of other combinations.

— MR. Cox ***1893***, *Cinderella: Three hundred and forty-five variants [**?**]*

The first web browser, Nexus, made its appearance in 1990 [**?**]. At its inception, web browsing consisted solely of retrieving and displaying small, static textual web pages. In simpler terms, users could only read page content without any interactive components. However, the escalating computing power of devices, the proliferation of the internet, the valuation of internet based companies and the demand for more engaging user experiences birthed the concept of executing code in conjunction with web pages. In 1995, the Netscape browser revolutionized this concept by introducing JavaScript [**?**], a language that allowed code execution on the client-side. Interactive web content immediately highlighted benefits: unlike classical native software, web applications do not require installation, are always up-to-date, and are accessible from any device with a web browser. Significantly, since the advent of Netscape, all browsers have offered JavaScript support. In the present day, the majority of web pages incorporate not only HTML but also JavaScript code, which is executed on client computers. Over the past several decades, web browsers have metamorphosed into JavaScript language virtual machines. They have evolved into intricate systems capable of running comprehensive applications, such as video and audio players, animation creators, and PDF document renderers, like the one displaying this document.

JavaScript is presently the most widely utilized scripting language in all contemporary web browsers [**?**]. However, it is not without limitations due to the inherent characteristics of the language. For instance, each JavaScript engine necessitates the parsing and recompiling of the JavaScript code, resulting in substantial overhead. In fact, just parsing and compiling JavaScript code consume

[0] Compilation probe time 2023/10/27 12:21:33

the majority of the load times of websites [**?**]. In addition to performance limitations, JavaScript also has security concerns [**?**]. A notable example of this is the lack of memory isolation in JavaScript, which allows extraction of information from other processes [**?**]. These issues led the Web Consortium (W3C) to standardize a bytecode for the web environment in 2015, which is the WebAssembly (Wasm) language. Hence, WebAssembly became the fourth official language for the web.

1.1 WebAssembly

WebAssembly was designed with a focus on speed, portability, self-containment, and security [**?**]. It allows for all programs to be compiled ahead-of-time from source languages like C/C++ and Rust. Third-party compilers create WebAssembly binaries, potentially including optimizations, as in the case of LLVM. The Wasm language defines its Instruction Set Architecture as an abstraction, similar to machine code instructions but independent of CPU architectures [**?**]. This design allows web browsers to compile quickly to target architectures through a fast, one-to-one translation process.

The versatility of WebAssembly extends not just to web browsers, but to backend scenarios as well. Previous research has demonstrated the advantages of utilizing WebAssembly as an intermediary layer, noting improved startup times and more efficient memory usage when compared to containerization and virtualization [**?** **?**]. In response to these findings, the Bytecode Alliance introduced the WebAssembly System Interface (WASI) in 2019 [**?** **?**]. WASI facilitated the execution of Wasm with a POSIX system interface protocol, thus enabling the direct execution of Wasm in the operating system. To underline the significance of this development, consider the words of Solomon Hykes, the former CEO of Docker, in a tweet about WebAssembly and WASI:

> *If WASM+WASI existed in 2008, we wouldn't have needed to created Docker. That's how important it is. Webassembly on the server is the future of computing. A standardized system interface was the missing link. Let's hope WASI is up to the task! [**?**]*

WebAssembly is praised for its security, especially for its design that prevents programs from accessing data beyond their own memory. However, there has been less focus on potential vulnerabilities and attacks within WebAssembly's own memory [**?**]. In addition, WebAssembly binaries may be inherently vulnerable due to flaws in their source code [**?**]. There are also significant risks from side-channel attacks, as demonstrated by various researchers [**?** **?** **?**]. These vulnerabilities are not limited to the web browser environment, as WebAssembly is also used in the backend. Yet, existing WebAssembly research mostly reacts

to existing vulnerabilities, leaving the potential for unidentified attacks. In this dissertation, we propose a proactive approach to enhance WebAssembly security through Software Diversification.

1.2 Software Diversification

Software Diversification is the process of finding, creating, and deploying program variants of a given original program [**?**] for the sake of security. Cohen et al. [**?**] and Forrest et al. [**?**] pioneered this field by proposing software diversification through code transformations. They proposed to produce variants of programs while preserving their functionalities, aiming to mitigate vulnerabilities. Since then, transformations aiming at reducing the predictability of observable behavior of programs have been proposed. For example, works on this direction proposed to diversify programs control flow [**?**], instruction set [**?**], or the system calls they use [**?**]. Several of these transformations can be combined to produce less predictable variants.

While previous works on software diversification demonstrated the removal of vulnerabilities, in all cases, it can be used as a preemptive solution. For example, if a vulnerability is present in one program variant, discovering and disseminating it will not affect other variants. Software diversification has been widely researched, yet, the field does not study its application to WebAssembly.

This dissertation presents toolsets, approaches and methodologies designed to enhance WebAssembly security proactively through Software Diversification. First, Software Diversification could expand the capabilities of already settled WebAssembly analysis tools by incorporating diversified program variants, making it more challenging for attackers to exploit any missed vulnerabilities. Generated as proactive security, these diversified variants can simulate a broader set of real-world conditions, thereby making WebAssembly analysis tools more accurate. Second, we noted that current solutions to mitigate side-channel attacks on WebAssembly binaries are either specific to certain attacks or need the modification of runtimes. Software Diversification could mitigate yet-unknown vulnerabilities on WebAssembly binaries by generating diversified variants in a platform-agnostic manner.

1.3 List of contributions

In the space of Software Diversification we make the following contributions.

C1 Experimental contribution: For each proposed technique we provide an artifact implementation and conduct experiments to assess its capabilities. The artifacts are publicly available. The protocols and results of assessing the artifacts provide guidance for future research.

Contribution	Research papers			
	P1	P2	P3	P4
C1 Experimental contribution	✓	✓	✓	✓
C2 Theoretical contribution	✓		✓	
C3 Diversity generation	✓	✓	✓	✓
C4 Defensive diversification	✓	✓	✓	
C5 Offensive diversification				✓

Table 1.1

C2 Theoretical contribution: We propose a theoretical foundation in order to improve Software Diversification for WebAssembly.

C3 Diversity generation: We generate WebAssembly program variants.

C4 Defensive Diversification: We assess how generated WebAssembly program variants could be used for defensive purposes.

C5 Offensive Diversification: We assess how generated WebAssembly program variants could be used for offensive purposes, yet improving security systems.

1.4 Summary of research papers

This compilation thesis comprises the following research papers.

P1: **CROW: Code randomization for WebAssembly bytecode.**
Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
Measurements, Attacks, and Defenses for the Web (MADWeb 2021), 12 pages
`https://doi.org/10.14722/madweb.2021.23004`

Summary: In this paper, we introduce the first entirely automated workflow for diversifying WebAssembly binaries. We present CROW, an open-source tool that implements software diversification through enumerative synthesis. We assess the capabilities of CROW and examine its application on real-world, security-sensitive programs. In general, CROW can create statically diverse variants. Furthermore, we illustrate that the generated variants exhibit different behaviors at runtime.

P2: **Multivariant execution at the Edge.**
Javier Cabrera-Arteaga,Pierre Laperdrix, Martin Monperrus, Benoit Baudry
Moving Target Defense (MTD 2022), 12 pages
`https://dl.acm.org/doi/abs/10.1145/3560828.3564007` **Summary:**

In this paper, we synthesize functionally equivalent variants of a deployed edge service. These variants are encapsulated into a single multivariant WebAssembly binary, When executing the service endpoint, a random variant is selected each time a function is invoked. Execution of these multivariant binaries occurs on the global edge platform provided by Fastly, as part of a research collaboration. We demonstrate that these multivariant binaries present a diverse range of execution traces throughout the entire edge platform, distributed worldwide, effectively creating a moving target defense.

P3: **Wasm-mutate: Fast and efficient software diversification for WebAssembly.**
Javier Cabrera-Arteaga, Nicholas Fitzgerald, Martin Monperrus, Benoit Baudry
Under review, 17 pages
`https://arxiv.org/pdf/2309.07638.pdf`

Summary: This paper introduces WASM-MUTATE, a compiler-agnostic WebAssembly diversification engine. The engine is designed to swiftly generate semantically equivalent yet behaviorally diverse WebAssembly variants by leveraging an e-graph. We show that WASM-MUTATE can generate tens of thousands of unique WebAssembly variants in mere minutes. Importantly, WASM-MUTATE can safeguard WebAssembly binaries from timing side-channel attacks, such as Spectre.

P4: **WebAssembly Diversification for Malware evasion.**
Javier Cabrera-Arteaga,Tim Toady, Martin Monperrus, Benoit Baudry
Computers & Security, Volume 131, 2023, 17 pages
Summary: WebAssembly, while enhancing rich applications in browsers,

also proves efficient in developing cryptojacking malware. Protective measures against cryptomalware have not factored in the potential use of evasion techniques by attackers. This paper delves into the potential of automatic binary diversification in aiding WebAssembly cryptojacking detectors' evasion. We provide proof that our diversification tools can generate variants of WebAssembly cryptojacking that successfully evade VirusTotal and MINOS. We further demonstrate that these generated variants introduce minimal performance overhead, thus verifying binary diversification as an effective evasion technique.

Thesis layout

This dissertation comprises two parts as a compilation thesis. Part one summarises the research papers included within, which is partially rooted in the author's licentiate thesis [**?**]. Chapter 2 offers a background on WebAssembly and the latest advancements in Software Diversification. Chapter 3 delves into our technical contributions. Chapter 4 exhibits two use cases applying our technical contributions. Chapter 5 concludes the thesis and outlines future research directions. The second part of this thesis incorporates all the papers discussed in part one.